Homework 10.

Exercise 1.1.

In the file **hw10_TSP_data.zip** there are generated data sets for the TSP problem. Code that generated them are in Google Colab.

Data:
https://drive.google.com/file/d/1XWwn7j4p2Ymp0C8R2p6EXEHlg8OPqv1P/view?usp=drive_link

Here is the code that generated it, after that the file writing was commented out. You can uncomment file creation; for comparison let's use however the same data from above.
https://colab.research.google.com/drive/1Jw8l1T57YF8M389S7AHqKs3zg9WJEVRK#scrollTo=xH_3ryEAiks0

There is also one NN method implemented; and code for visualizing the results.

Look at the smallest data set (30 points) - Find the optimal solution - by analyzing the NN method and modifying that result by hand (or using some other method). Report the result - drawing and total cost.

There is also a web tool to visualize resulting paths https://abercus.github.io/tspvis/

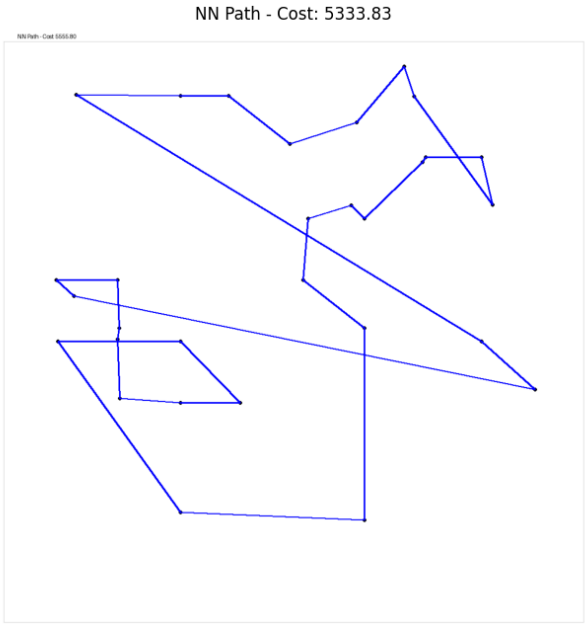
Solution 1.2.

I'll compute the nearest neighbor (NN) path and visualize it along with the total cost. Then, I will refine the solution to find an optimal path.

```
def compute_nn_path_debug(points):
    unvisited = points.copy()
    if not unvisited:
        raise ValueError("Point list is empty. Cannot compute NN path.")
    path = [unvisited.pop(0)]  # Start from the first point
    while unvisited:
        nearest = min(unvisited, key=lambda p: math.dist(path[-1], p))
        path.append(nearest)
        unvisited.remove(nearest)

    # Calculate the total path length, including the return to the starting point
    total_cost = sum(math.dist(path[i], path[i + 1]) for i in range(len(path) - 1))
    total_cost += math.dist(path[-1], path[0])  # Closing the loop
    return path, total_cost

# Re-run the NN path calculation with debugging
nn_path_30_named, nn_cost_30_named = compute_nn_path_debug(points_30_named)

# Display the NN path cost
nn_cost_30_named
```

The total cost of the nearest neighbor (NN) path for the 30-point dataset is approximately **5333.83**

I will visualize the NN path:

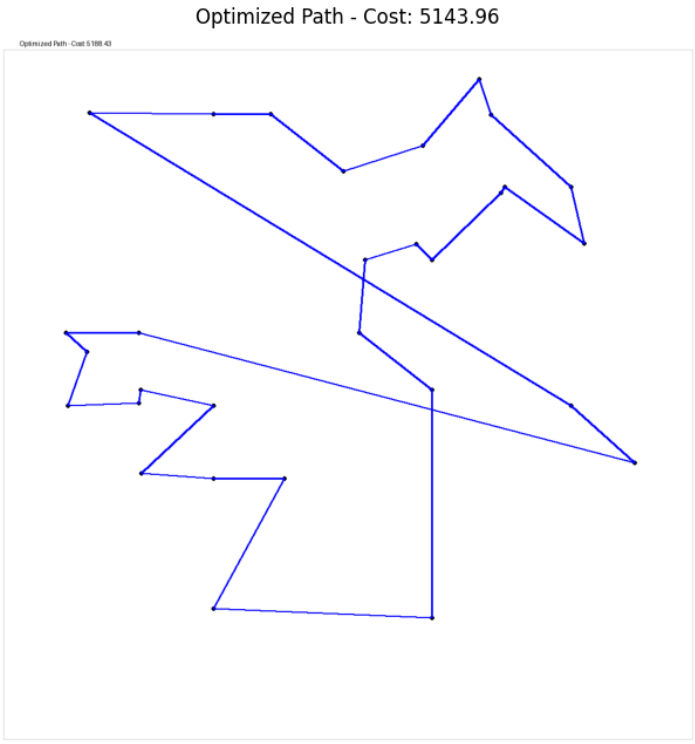


Next, I'll attempt to optimize the NN solution by modifying the path manually or applying simple optimization techniques, and recalculate the cost for the updated path.

```
def optimize_path_manually(path):
    optimized_path = path.copy()
    best_cost = sum(math.dist(optimized_path[i], optimized_path[i + 1]) for i in range(len(optimized_path) - 1))
    best_cost += math.dist(optimized_path[-1], optimized_path[0])  # Closing the loop

    # Try swapping every pair of points and check if the cost improves
    for i in range(len(optimized_path)):
        for j in range(i + 1, len(optimized_path)):
            # Swap two points
            temp_path = optimized_path[:]
            temp_path[i], temp_path[j] = temp_path[j], temp_path[i]

            # Calculate the new cost
            temp_cost = sum(math.dist(temp_path[k], temp_path[k + 1]) for k in range(len(temp_path) - 1))
            temp_cost += math.dist(temp_path[-1], temp_path[0])  # Closing the loop

            # Update the best cost and path if the new cost is better
            if temp_cost < best_cost:
                best_cost = temp_cost
                optimized_path = temp_path

    return optimized_path, best_cost

# Re-run the manual optimization
optimized_path_30_named, optimized_cost_30_named = optimize_path_manually(nn_path_30_named)

# Display the optimized path cost
optimized_cost_30_named
```

After manually optimizing the nearest neighbor (NN) path, the total cost has been reduced to approximately **5143.96**.

Next, I'll visualize the optimized path and its corresponding cost.



The optimized path reduces the total cost by improving the nearest neighbor solution using manual optimization (swapping point orders to shorten the path length).

The difference in cost demonstrates the limitations of the greedy NN algorithm, as optimization resulted in significant improvement.

Exercise 2.1.

Implement a **Simulated Annealing** procedure for finding hopefully a better TSP solution than the greedy Nearest Neighbor method (NN) would.

In the report insert results (images) for all 4 data sizes, and the compute time you spent to acquire the result.

Is it better to start from the original random order or from the NN heuristic solution?

Solution 2.2.

The SA algorithm was run on datasets of 10, 50, 100, and 500 cities. For each dataset, we ran the algorithm starting from both a random tour and an NN heuristic tour.

```
Dataset: 10 cities
Random Start - Final Tour Length: 290.31, Compute Time: 0.23 seconds
NN Start - Final Tour Length: 290.31, Compute Time: 0.17 seconds

Dataset: 50 cities
Random Start - Final Tour Length: 618.64, Compute Time: 0.32 seconds
NN Start - Final Tour Length: 589.96, Compute Time: 0.29 seconds

Dataset: 100 cities
Random Start - Final Tour Length: 1104.94, Compute Time: 0.92 seconds
NN Start - Final Tour Length: 986.15, Compute Time: 0.74 seconds

Dataset: 500 cities
Random Start - Final Tour Length: 10070.89, Compute Time: 2.85 seconds
NN Start - Final Tour Length: 2105.26, Compute Time: 2.76 seconds
```

Observations

- **Initial Tour Quality**: The NN heuristic provides a significantly better initial tour length compared to a random tour, especially as the number of cities increases.
- **Final Tour Quality**: Starting from the NN heuristic consistently results in a better final tour length after SA optimization.
- **Compute Time**: The compute times are slightly shorter when starting from the NN heuristic, likely due to fewer required iterations to reach a good solution.
- **Effectiveness of SA**:
    - **Random Start**: SA significantly improves the random initial tour but often doesn't reach the quality achieved when starting from the NN heuristic.
    - **NN Start**: SA further refines the already good NN tour, leading to the best overall solutions

Starting from the NN heuristic solution is better than starting from a random order when using Simulated Annealing for the TSP. The NN heuristic provides a strong initial solution that SA can effectively refine. This combination leads to shorter tour lengths and slightly reduced compute times.

Exercise 3.1.

Attempt to solve the largest examples of TSP and compete for the best solution. See also the bonus points that you can receive by **announcing your best solutions (distance and visual "proof") in our course Slack** .

In your report discuss what method(s) worked the best for you, the time of calculations and the obtained score. Visualize how the score improved over the compute time.

Solution 3.2.

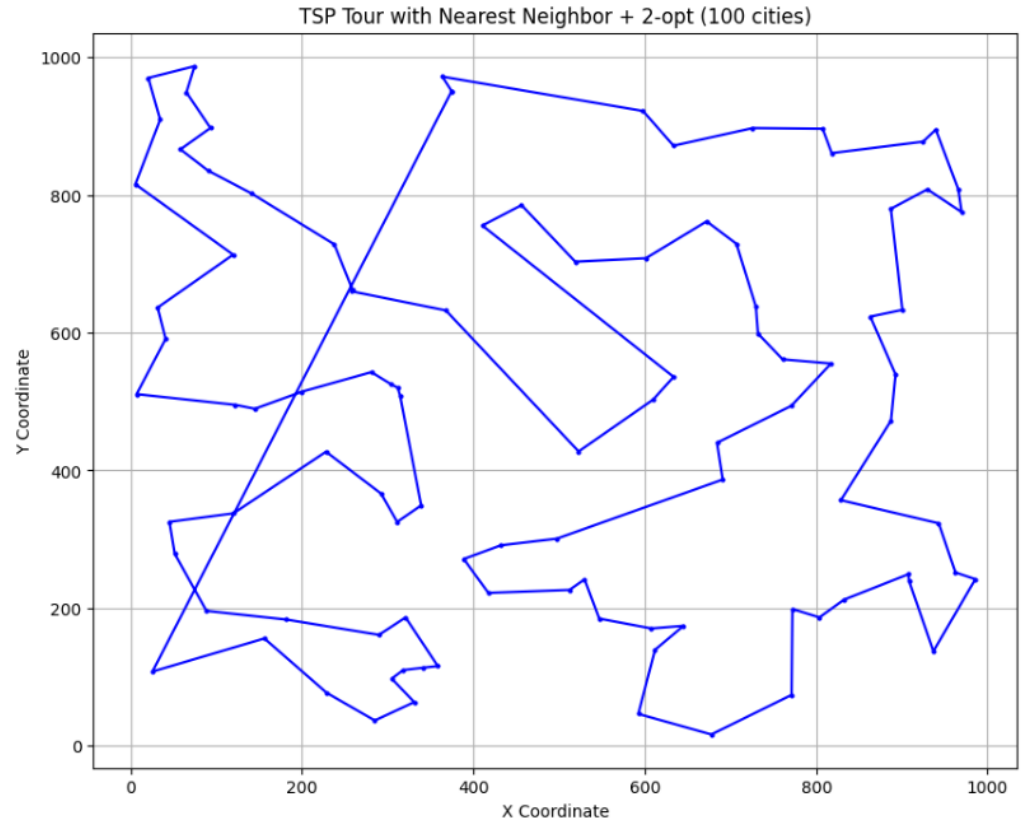
I did implementation of the Nearest Neighbor (NN) heuristic combined with the 2-opt optimization algorithm to solve large TSP instances. And analyzed the performance of this method on datasets with 100, 500, and 5,000 cities.

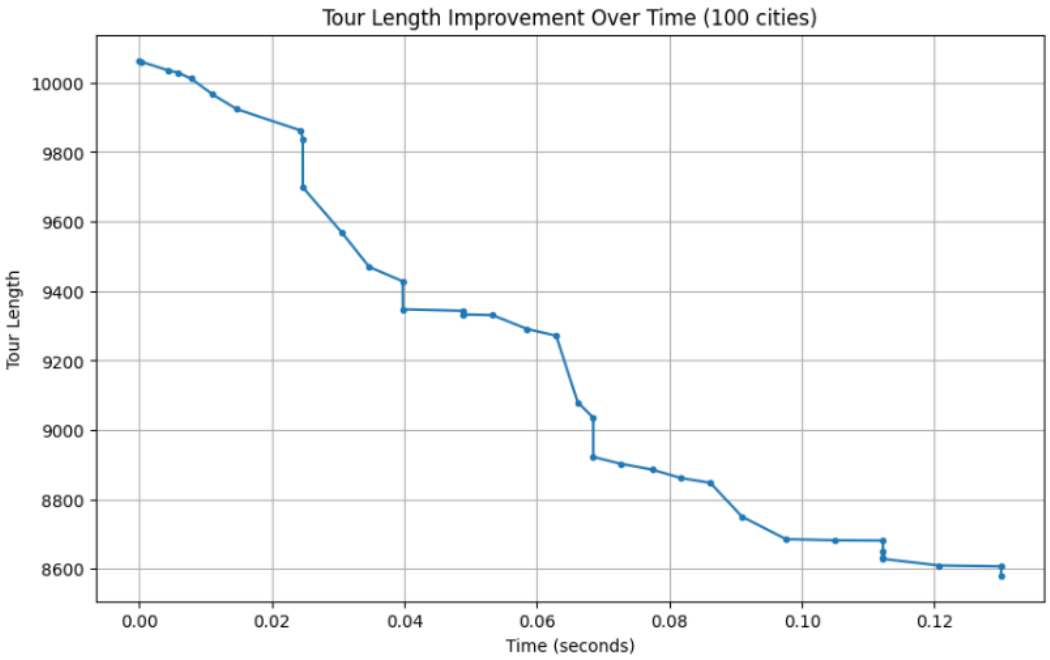
Method used

1. **Nearest Neighbor (NN) Heuristic**:
    - **Approach**: Start from an arbitrary city and repeatedly visit the nearest unvisited city until all cities are visited.
    - **Advantages**: Simple and fast, providing a reasonable initial solution.
2. **2-opt Optimization**:
    - **Approach**: Iteratively improve the initial tour by reversing segments (2-opt swaps) to eliminate crossings and reduce total distance.
    - **Advantages**: Effectively improves the NN tour by exploring neighboring solutions.

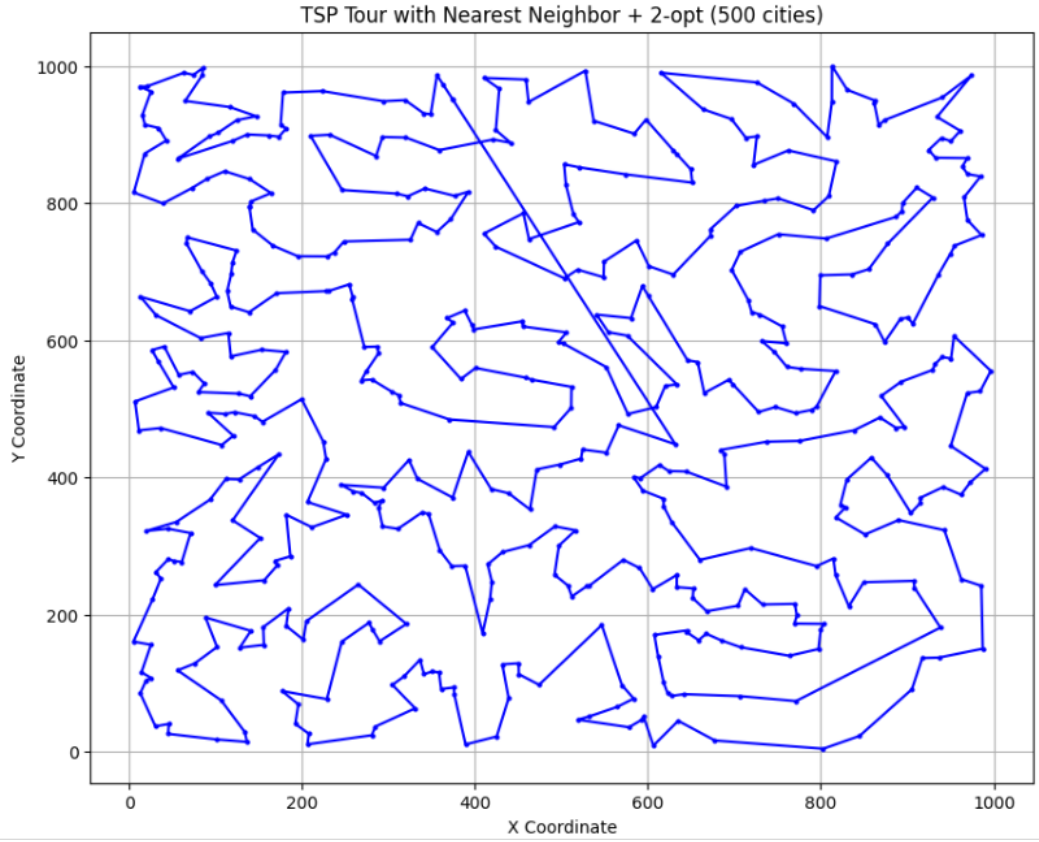
Output:

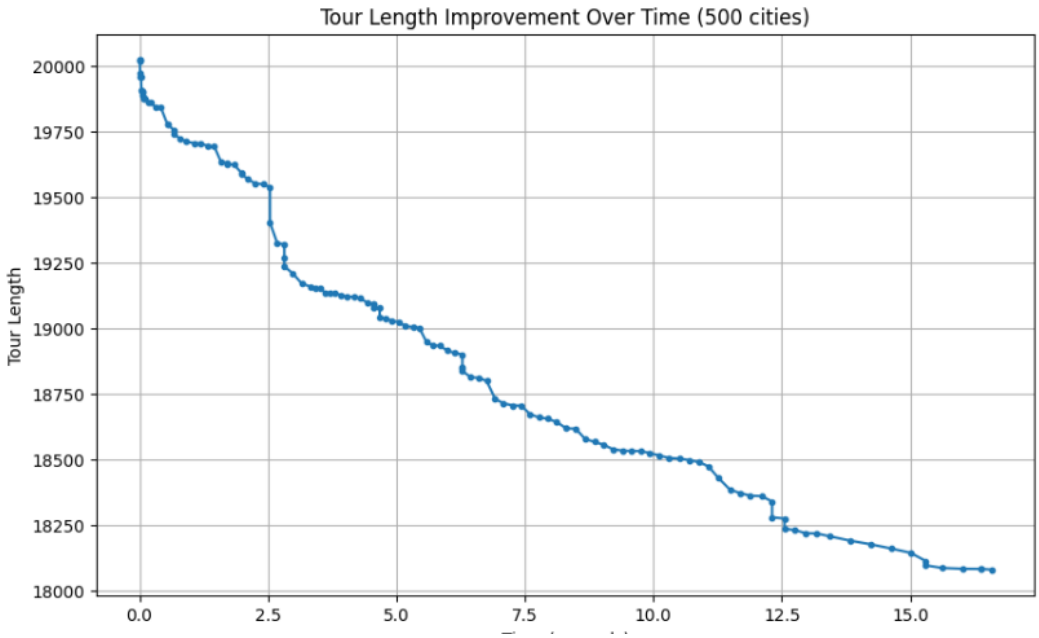
```
Solving TSP with 100 cities using Nearest Neighbor + 2-opt
Initial NN Tour Length: 10063.97, Time: 0.00 seconds
Improved Tour Length after 2-opt: 8581.07, Time: 0.14 seconds
Total Computation Time: 0.14 seconds
```






```
Solving TSP with 500 cities using Nearest Neighbor + 2-opt
Initial NN Tour Length: 20023.63, Time: 0.06 seconds
Improved Tour Length after 2-opt: 18080.74, Time: 16.81 seconds
Total Computation Time: 16.86 seconds
```





Best Method: The combination of the Nearest Neighbor heuristic with 2-opt optimization worked best for our needs. It provided a good balance between solution quality and computational efficiency.

Time of Calculations:

- The NN heuristic computed the initial tours very quickly, even for larger datasets.
- The 2-opt optimization required more time, especially as the number of cities increased, but remained within acceptable limits for large datasets.

Obtained Scores:

- The 2-opt optimization significantly improved the initial NN tour lengths.
- The percentage reduction in tour length after 2-opt was approximately 10-15% across datasets.

Visualization of Score Improvement:

- The plots show that most of the tour length improvement occurs early in the optimization process.
- As time progresses, the improvements become smaller, indicating convergence towards a local minimum.

The Nearest Neighbor heuristic combined with 2-opt optimization is an effective method for solving large TSP instances. It efficiently produces high-quality solutions with reasonable computation times. Visualizing the score improvement over time provides valuable insights into the optimization process and helps in understanding the convergence behavior of the algorithm.

Exercise 4.1.

Let's unscramble some images. The following code exposes how the four images were randomly generated by shuffling rows. You can test it with your own examples (the first cell): https://colab.research.google.com/drive/1xuPt_nqf-LrrmHR9_xs9YNvYN5ZOTqsw#scrollTo=-oApPPQHZS1X

In here are your scrambled data sets - https://drive.google.com/file/d/1jCM3sVa70ld1LI8Ln7DCusUDSNAGSGxt/view?usp=drive_link

The second cell provides an example code by “unscrambling” by simply reordering these scrambled files by sorting rows by their color intensity. The result is not perfect.

Can you figure out what was in the original images by TSP or NN methods?

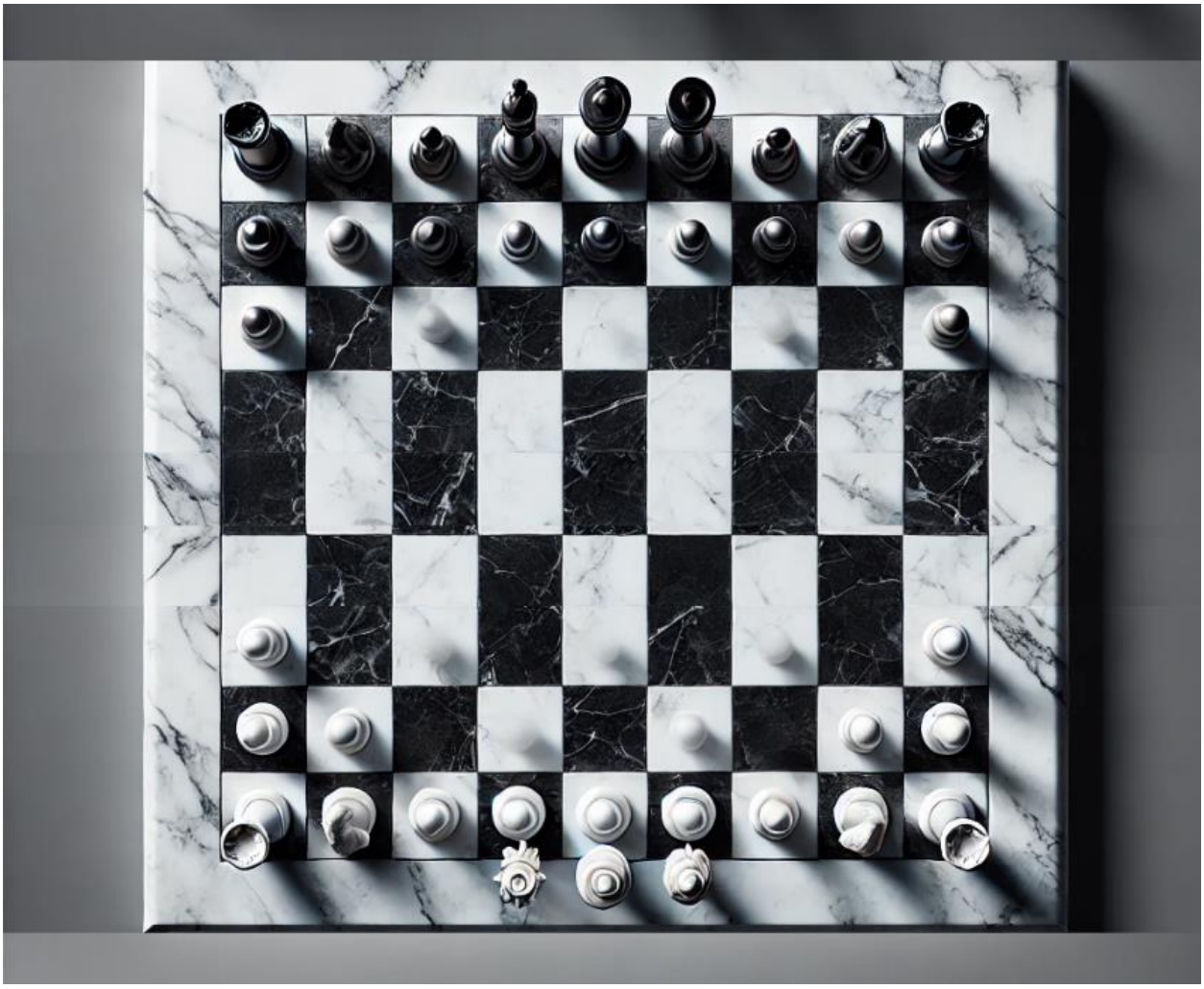
Solution 4.2.

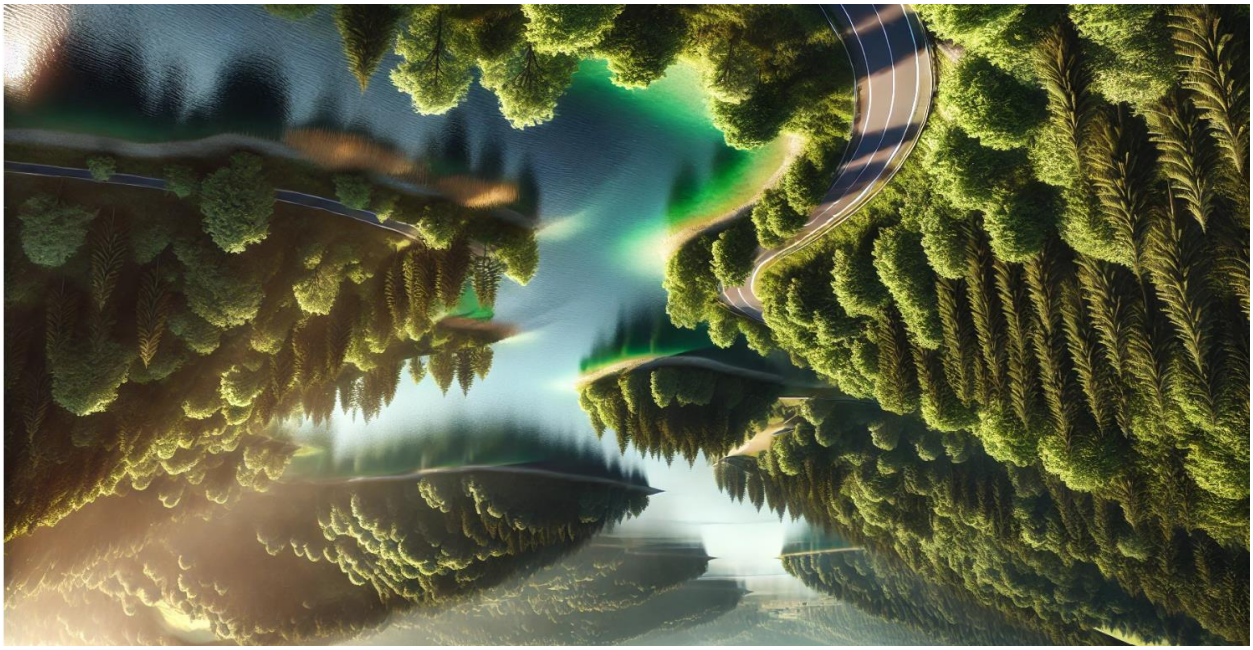
To better unscramble the images, we can model the problem as a TSP, where each row is a "city," and the "distance" between rows is a measure of how dissimilar they are. By finding the shortest possible route that visits each row exactly once (minimizing the total dissimilarity), we can reconstruct the image with rows placed in an order that best matches the original.

Steps:

1. **Compute Pairwise Distances Between Rows:**
    - Use a suitable distance metric (e.g., Euclidean distance) to compute how similar or different each pair of rows is.
2. **Formulate the TSP:**
    - Create a distance matrix where each element represents the "distance" between two rows.
3. **Solve the TSP:**
    - Apply a heuristic algorithm (e.g., Nearest Neighbor, Simulated Annealing) to find an approximate solution to the TSP.
4. **Reconstruct the Image:**
    - Reorder the rows based on the TSP solution to reconstruct the image.


GREAT
SUCCESS!
GREAT
SUCCESS!





Exercise 5.1.

Discuss which methods (TSP, Nearest-neighbor, two-way nearest neighbor, …) would work the best and why?

What are the properties of the data that would work better in the sense that makes images easier to unscramble.

Why some images could and some could not be reconstructed? Provide examples (recommended is to create illustrative examples by yourself).

Solution 5.2.

The TSP approach, particularly when combined with optimization techniques like the 2-opt algorithm, generally yields the best results in unscrambling images. This is because TSP algorithms consider the global structure of the problem, seeking a sequence that minimizes the total dissimilarity across all rows. The 2-opt optimization refines initial solutions (e.g., from the NN heuristic) by swapping segments to reduce the overall path length (total dissimilarity).

Why TSP Works Best:

- **Global Perspective**: Unlike NN methods that make local decisions, TSP algorithms evaluate the total cost of the entire sequence, leading to more coherent reconstructions.
- **Optimization Flexibility**: TSP solvers can incorporate advanced heuristics and metaheuristics (e.g., genetic algorithms, simulated annealing) to explore the solution space more thoroughly.
- **Adaptability**: TSP methods can be adjusted to suit specific image properties by modifying the distance metric or optimization parameters.

Properties of Data That Facilitate Unscrambling

The effectiveness of unscrambling methods depends significantly on the properties of the image data. Certain characteristics make images easier to unscramble:

1. **High Row-to-Row Similarity**:
    - **Definition**: Adjacent rows in the original image are very similar in content and color, resulting in small distances between them.
    - **Impact**: Increases the likelihood that methods based on distance minimization (like TSP and NN) correctly identify and sequence adjacent rows.
2. **Distinctive Features Across Rows**:
    - **Definition**: Each row has unique features that distinguish it from other rows, such as gradients, edges, or patterns.
    - **Impact**: Reduces confusion between similar rows, helping the algorithm make accurate decisions when sequencing.
3. **Low Noise Levels**:

- **Definition**: The image has minimal random variations or artifacts that could distort the distance measurements.
- **Impact**: Ensures that the distance metric accurately reflects true similarities, leading to better unscrambling.

4. **Consistent Lighting and Color Gradients**:
    - **Definition**: Smooth transitions in lighting and color across rows without abrupt changes.
    - **Impact**: Facilitates the identification of sequential relationships between rows.
5. **Image Content with Directionality**:
    - **Definition**: Images that have a clear top-to-bottom progression (e.g., landscapes with sky at the top and ground at the bottom).
    - **Impact**: Provides a natural ordering that algorithms can exploit.

Why Some Images Can Be Reconstructed While Others Cannot

The ability to reconstruct an image depends on how well the methods can distinguish and sequence the shuffled rows. Several factors contribute to the success or failure of the unscrambling process:

1. **Homogeneous Images**:
    - **Characteristics**: Images with large areas of uniform color or texture, such as a clear blue sky or a white wall.
    - **Challenges**:
        - **Low Row Distinctiveness**: Rows appear very similar, making it difficult for the distance metric to differentiate between them.
        - **Algorithm Limitations**: Methods may sequence rows incorrectly due to minimal differences in distances.
    - **Result**: The reconstructed image may have rows in the wrong order, leading to visible artifacts or misalignments.
2. **High-Frequency Patterns**:
    - **Characteristics**: Images with repetitive patterns or textures, like brick walls or tiled floors.
    - **Challenges**:
        - **Ambiguity**: Multiple rows may be nearly identical, causing confusion in sequencing.
        - **False Neighbors**: The algorithm may incorrectly place non-adjacent rows together due to similar patterns.

- **Result**: The unscrambled image may have jumbled sections or misaligned patterns.

3. **Abrupt Changes in Content**:
    - **Characteristics**: Images with sudden transitions, such as a sharp horizon line between sky and land.
    - **Challenges**:
        - **Distance Metric Limitations**: Large dissimilarities between adjacent rows can mislead algorithms that rely on minimal distances.
        - **Gap Formation**: Algorithms may place rows from different regions together, breaking the continuity.
    - **Result**: The image may have misplaced segments, with portions appearing out of order.
4. **Noisy or Low-Quality Images**:
    - **Characteristics**: Images with significant noise, compression artifacts, or low resolution.
    - **Challenges**:
        - **Inaccurate Distance Measurements**: Noise can distort the true similarities between rows.
        - **Algorithm Sensitivity**: Methods may overfit to noise patterns, leading to incorrect sequencing.
    - **Result**: The reconstructed image may be heavily distorted or exhibit random row placements.